



REFERENCE MANUAL

HEWLETT-PACKARD PRIVATE

ALPHA COMPUTER SYSTEM

and Central Processor Module

HEWLETT  PACKARD

Do Not Reproduce 42

FOREWORD

This section (IV) is the only one written as of this date. Ultimately the manual will consist of ten sections which will be released one at a time as they are completed. Major topics not discussed in this section are I/O, interrupts, and instruction definitions. For now the ERS must be referred to for this information. Although the format looks somewhat final this material is no more than a first draft. Changes can and will be made before final review around the end of this year. The double column format is used primarily for reproduction economy. Due to time considerations, we do not plan to update this draft until just before final review.

One anticipated change, decided upon just prior to this printing, is that the terms primary memory and secondary memory will be used in place of the restrictive terms core memory and disc memory. This will lessen or delay the chances of documentation obsolescence due to the coming new memory technologies.

SECTION IV

of

Alpha Reference Manual
—System and Central Processor—

Table of Contents

IV MEMORY SEGMENTATION

Introduction	4-1
Reserved Core	4-3
System Library	4-3
Operating System	4-3
Compilers	4-5
User Allocations	4-5

Code and Data Segments	4-5
Segments in Memory	4-6
Code Segments	4-7
Data Segments	4-9
Examples of Stack Operation	4-14
Basic Arithmetic	4-14
Procedure Calls	4-15
Recursion	4-18

INDEX

Terms Defined in Section IV

absent	4-1	parameter passing	4-15
actual parameters	4-15	privileged data area	4-10
allocate	4-2	procedure	4-7
BEGIN statement	4-16	procedure body	4-16
code segment	4-1	procedure call	4-7
code segment number	4-7	procedure declaration	4-16
Code Segment Table	4-2	procedure head	4-16
Code Segment Table Pointer	4-3	procedure name	4-16
Current Process Control Block pointer	4-9	procedure parameters	4-13
data domain	4-5	procedure statement	4-16
data segment	4-2	process	4-5
Data Segment Table	4-2	Process Control Block	4-5
deallocate	4-2	Process Control Block Extension	4-10
delta Q	4-13	program	4-5
END statement	4-16	recursive procedure	4-18
external program label	4-7	re-entrant	4-1
file area	4-2	relocatable binary module	4-2
formal parameters	4-15	segment	4-1
free space links	4-6	Segment Transfer Table	4-7
global data area	4-12	stack	4-11
global variable	4-16	stack marker	4-13
implicit addressing	4-15	swapping area	4-1
job	4-5	system library	4-3
Job Main Process	4-5	temporary storage	4-14
local program label	4-7	top-of-stack	4-11
local variables	4-14	uncallable bit	4-9
main memory links	4-6	value part	4-16
operating system	4-3	virtual memory	4-1

SECTION IV

MEMORY SEGMENTATION

In most instances, the user need never be aware of the structural details to be presented in this section. Memory addressing is accomplished automatically by features of the hardware under control of a software operating system, which in turn serves a language compiler. Thus the user is normally several levels removed from the intricacies of the hardware.

The primary intent of this section is to illustrate the power and flexibility built into the system. But additionally, this section also serves as an introduction to the subject of memory segments for those persons, such as interface designers, operating system programmers, and maintenance personnel, who have a requirement for such knowledge.

INTRODUCTION

To begin with, it is assumed that the reader is familiar with the system characteristics—that is, as viewed externally by the user through standard software. It is the purpose of this introduction to provide a bridge from the overall “system” viewpoint into the functionings of the hardware, as regards memory operations. Therefore no attempt will be made to explain the concepts of jobs and processes, any more than is necessary for the following discussions. The reader should refer to separate documentation for the software systems, if full definitions of these concepts are required.

First it is necessary to establish what is meant by *virtual memory*. As shown in figure 4-1, virtual memory consists of the main core memory plus an area of mass storage called the *swapping area*. The swapping area, typically on disc or drum memory, consists of a collection of pieces of code or data, defined as segments, which are not presently in core but which may be called in by the executing programs. A *segment* is the basic entity for transfers between core memory and the swapping area. Whether a segment is in core or *absent* (on disc), it is nevertheless part of the virtual memory. From the point of view of the user, he is working with a memory that appears to be many times larger than actual core size. In fact, his own program may exceed the 65K-word maximum of core capacity, and still allow room for many other users on the same machine.

At this point the reader should be visualizing a dynamic situation in which various segments are being swapped rapidly between core memory and the swapping area of disc memory, according to the demands of the executing programs. Also bear in mind that several users may be on the machine at a given time, and that each user may have several segments.

Now the questions arise: where did the segments come from (i.e., how were they created), and how are they eventually eliminated? To answer these questions it is necessary to understand that there are two distinct types of segments, code segments and data segments. Thus there are two methods of origin. See figure 4-2.

A *code segment* consists entirely of information that is not subject to change during program execution. This includes the instructions of the program itself, constants, and an area for interprocedure links. No modifiable data may be interspersed with the instructions in a code segment, and in no way is it possible to write into or alter a code segment (or its formative parts) once it has been compiled. It is this feature which allows code to be *re-entrant*, meaning that a given sequence of instructions can be in simultaneous use by several users — or, can be entered several times by the same user, whether or not preceding entries are concluded. An example at the end of this section (Recursion) will illustrate a procedure which, after being entered by the main program, will call itself several times before any exit is given.

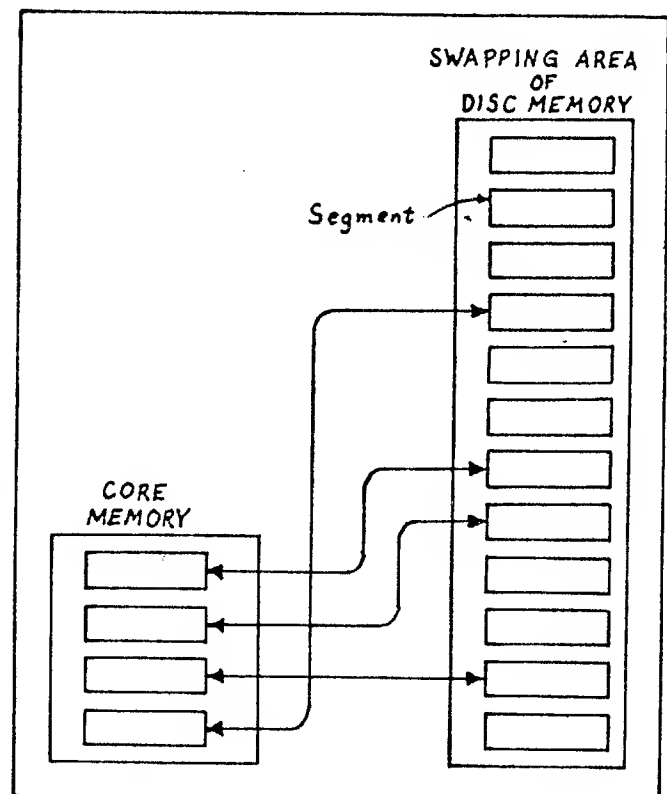


Figure 4-1. Virtual Memory

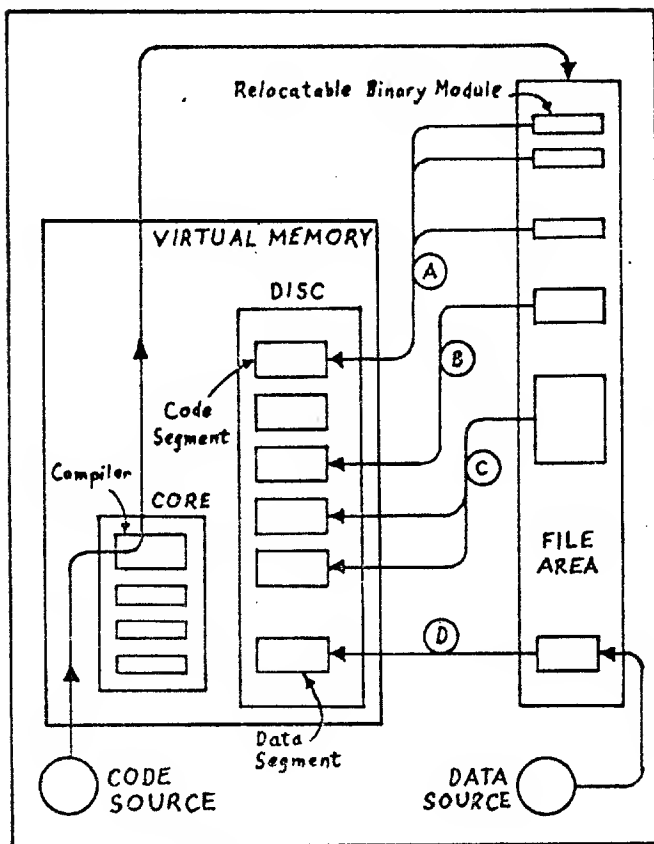


Figure 4-2. Sources of Segments

As shown in figure 4-2, code is entered into the computer in some source language, is translated to binary form by a compiler, and is stored in the *file area*. The file area is strictly a storage or holding area, and instructions are not executable from here. This area may be on the same mass storage device as the swapping area, but may also be a separate unit such as a slower disc — since speed of transfer is not as crucial here as it is for dynamic swapping of segments.

Each compiled program or subprogram exists in the file area as a *relocatable binary module*. When the user is ready to execute his program, the appropriate command is given and the operating system loads the binary modules of his program into the swapping area of virtual memory. Simultaneously with this transfer, the binary modules are formed into segments. This occurs in one of three ways, illustrated by examples A, B, and C in figure 4-2. The choice depends on what commands the user may have given. In example A, the user has specified three binary modules to be combined into one segment. In example B, one module forms one segment. In example C, the module is split into two segments; in this case, the user may have specified a break-point, or it may have been left to the discretion of the loader. Again, these are matters of programming and the methods of specifying a segment are of no great significance in the present context.

In any case, to transfer a code segment into virtual memory, in the manner indicated above, is to *allocate* that

segment. Every allocated code segment has an entry in the *Code Segment Table*, which is a set of reserved locations in core memory that tells both the hardware and the software exactly where each code segment is located. The table lists a memory address if the segment is core resident, or a disc address if disc resident, plus the segment length. It is maintained by the operating system.

A *data segment* consists only of data. Like the code segment, a data segment is fully protected. No user (more strictly, no process) may have access to the data segment of another user (or process). Generally speaking, each process defined by a user causes a data segment to be created. Initially, when the code segments are allocated, the data segment contains no actual data, but consists only of an initial stack having some initializing information. (Stack is defined later.) But at least the data segment is allocated — that is, a place for data is established.

Like code segments, data segments have entries in a table, called the *Data Segment Table*, which keeps track of where each data segment is located. Unlike the Code Segment Table, however, the Data Segment Table is not maintained in reserved core; its location is known only to the operating system software.

Once all segments have been allocated, the operating system transfers into core memory the code segment containing the entry point of the first process in the program, and execution begins. The data segment will also be present in core at this time.

As execution progresses, data will enter and leave the data segment — perhaps as the result of various computations, or perhaps via an external data source (example D in figure 4-2).

Eventually the last instruction in a given process will be executed. At that time the operating system will *deallocate* all segments associated exclusively with that process. That is, they will lose their entries in the Code Segment Table and the Data Segment Table, and the respective code and data will be overlaid by other segments coming into the system. For a time, of course, the old code and data will physically continue to exist in the virtual memory, but there is no means by which this information can be retrieved. Thus if there is some information to be saved as the result of process execution, the process itself must save such information in the file area.

Referring back to figure 4-1, the reader should at this point be able to visualize not only the swapping of segments in and out of core, but also the creation and elimination of various segments as new user processes come into the system and other processes come to an end. Obviously the areas occupied by segments in both core memory and disc memory will dynamically shrink and expand according to demands placed on the system. (To maintain optimum efficiency, the operating system has a timer and method of keeping usage statistics, so that the less important or less frequently used segments are most eligible for temporary swapping out to the disc.)

Now that the basic concept of a segment has been introduced, it is possible to show how the segment fits into the overall scheme of things.

Figure 4-3 is an overview of the major system elements. This figure shows the software that might exist in the hardware at a given instant of time. It does not attempt to show the possible links between elements, nor the relationships that can exist among various processes. It is simply a snapshot view of elements, showing location and constitution.

Note that the software exists either (or both) in core memory or in mass storage, here assumed to be disc. Note also that when a user's information is on disc (both code and data), it can be either in the file area or, after being formed into segments, in the swapping area.

The following paragraphs describe each of the elements shown in figure 4-3.

RESERVED CORE

Only 12 memory locations are "reserved" in the strictest sense — i.e., having a known, fixed address. These are the first 12 addresses. See table 4-1. In addition, however, there is also a permanent table which is reserved in the sense that, once established, each entry has a permanent allocation. The upper limit of the table, however, is flexible, depending on how many entries there are in the table. This table is the Device Reference Table (to be defined and discussed in a later section). It begins at octal location 14 and uses four locations for each device existent in the system.

The 12 fixed memory allocations can be divided into four groups, the first of which is only a single location, address 0. This location contains the *Code Segment Table Pointer*, which is the absolute address of the first entry in the Code Segment Table. Since this table may be moved by the operating system to any convenient place in memory at any time, the easily referenced location of address 0 makes the current location of the table readily known to both hardware and software.

The second group of fixed memory allocations, addresses 1, 2, and 3, is used for cold load operation. Location 1 will contain the initial P-register value, and locations 2 and 3 will be used by the cold load program during execution.

The third and fourth groups each apply to separate processors, if a dual-processor system is used. Locations 4 through 7 provide a Current Process Control Block pointer, two interrupt stack pointers, and an interrupt reference counter for processor 1. Octal locations 10 through 13 provide the same for processor 2. The Current Process Control Block pointers will be discussed in this section under the heading "Data Segments", and the interrupt stack pointers and counters will be discussed in the section on interrupt processing.

Table 4-1. Fixed Memory Allocations

LOCATION	CONTENTS
0	Code Segment Table Pointer
1	P-Register Cold Load Value
2	For Cold Load
3	For Cold Load
4	CPCB Pointer 1
5	QI 1
6	ZI 1
7	IR 1
10	CPCB Pointer 2
11	QI 2
12	ZI 2
13	IR 2
14	<div> <div></div> <div>First Entry</div> <div>Device Reference</div> <div>Table</div> <div>(Device #3)</div> </div>
15	
16	
17	

SYSTEM LIBRARY

The *system library* is a flexible means of sharing frequently used routines among many users. In addition to standard library routines, the user may enter and delete routines of his own in the library.

A library routine might be one procedure in a segment, a whole segment, or a set of segments. As shown in figure 4-3, some segments which contain (or are a part of) certain library routines are permanently allocated. That is, they have entries in the Code Segment Table. Other library segments remain in the file area until such time that a user makes a request for one of its routines. At that time the operating system will load the affected segments, create entries in the Code Segment Table, and provide appropriate links for the user to access the desired routine.

OPERATING SYSTEM

The *operating system* is the master supervisory program, overseeing the allocation of memory, controlling the loader, swapping user segments in and out of core, designating time to individual users, and so on.

In this manual, no particular version or configuration of an operating system is implied, but rather the usage of the term is meant to apply in a general sense to any operating system used with this computer.

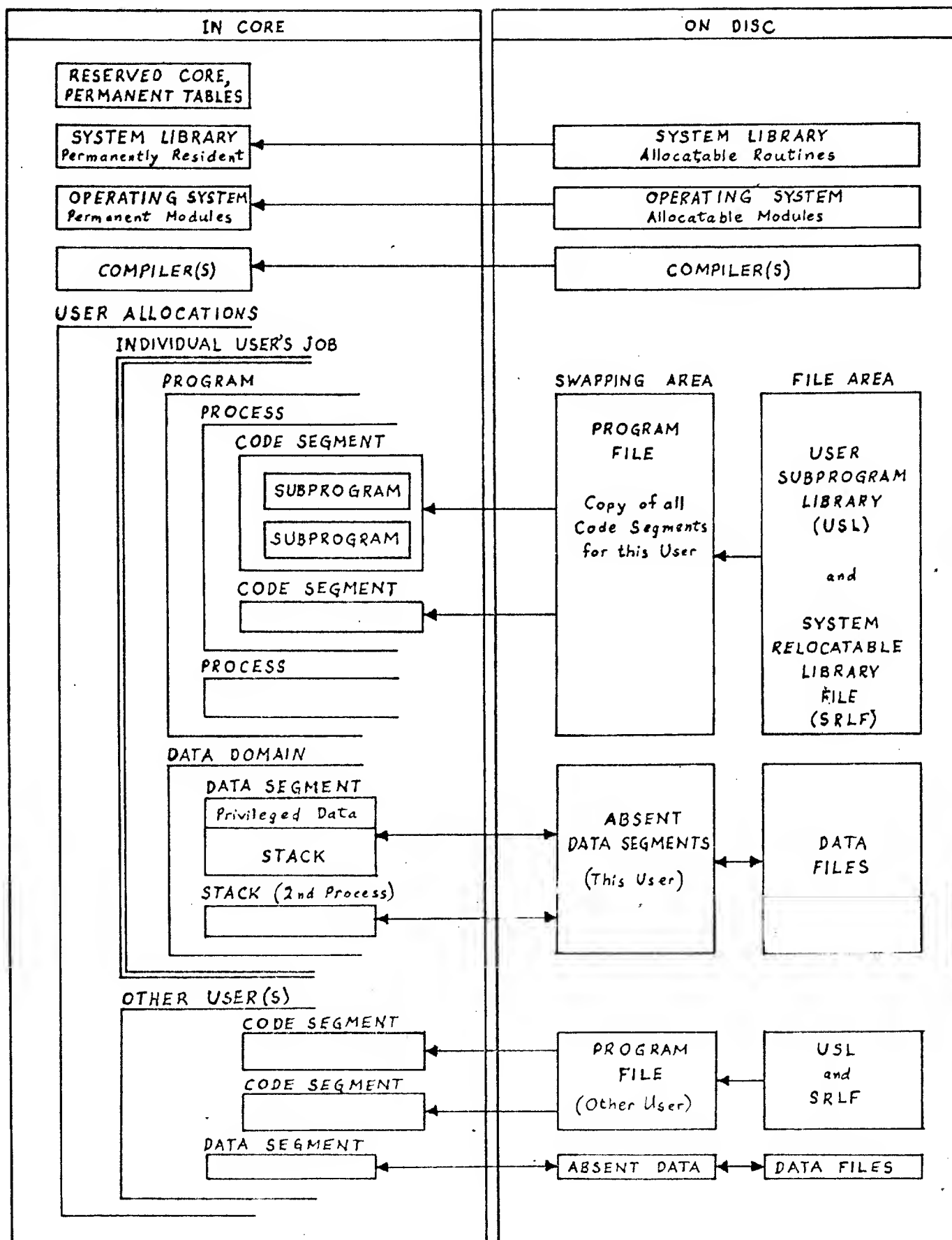


Figure 4-3. Software Elements in Hardware

As indicated in figure 4-3, not all parts of the operating system need to be permanently resident in core. Certain modules may be retained in the file area, and be allocated on a requirement basis.

COMPILERS

Several language compilers are available. If only one compiler is used on a system, it might remain permanently resident. For a multicompile system, however, it is more efficient to retain the permanent copy of each compiler in the file area, and to copy the compiler into core memory only when required. Due to the re-entrant feature for all programs run on this computer, only one copy of a compiler needs to be present in core, regardless of how many users may be simultaneously compiling. The operating system keeps a count of how many users are using a given compiler, and when this count reaches zero, the compiler is deallocated.

USER ALLOCATIONS

As shown in figure 4-3, the remaining space (after allocations for reserved core, library, operating system, and compilers) is available for users. This space includes both core space and disc space. Bear in mind that the relative block sizes in figure 4-3 do not indicate comparative sizes of space; the file and swapping areas, for example, may be many times the size of any allocation in core memory.

USER JOB. When a user logs into the system, he establishes a *job*. During the course of his job he will execute a *program* upon information contained in a separate and distinct *data domain*. A program embraces all code which may come into the machine during the course of a job. Similarly, the user's data domain consists of all data that is used or generated during the course of a job.

PROCESS. A program is executed on the basis of individual processes. A *process* cannot be strictly defined as a physical entity (e.g., that it contains a certain number of code segments). Rather, a process should be viewed in a dynamic sense, as a "window" across the system, embracing a varying set of segments that come and go, typically including segments from the system library.

During the course of a process, various code segments will necessarily become allocated, and it may even create other processes which will in turn allocate still other code segments. When at last a process comes to an end, all of the segments it has (exclusively) caused to be allocated will become deallocated.

An important feature of a process is that each process created causes one data segment to be created. The data segment created by a process is for the exclusive use of that process (though the operating system does provide for communication of data between related processes). Note in figure 4-3 that two data segments are shown in the data domain, corresponding to the two shown processes.

As implied earlier, the operating system keeps track of the various processes existing in the system. It does this through an interdependence or relationship between processes, whereby information embedded in each main process tells what other processes are currently existing (or "descendant") on that particular branch of a tree structure, and where they are located. Each *Job Main Process* (the "ancestor" process for a given job) maintains this information in a *Process Control Block* table, and a "progenitor" process in the operating system keeps track of each *Process Control Block*.

Processes will again be mentioned under the headings of "Code Segments" and "Data Segments", but further details regarding their relationships, substates, priorities, means of data communication, queuing, dispatching, etc., are extraneous to the present discussion. Refer to the operating system documentation for this type of information.

CODE SEGMENT. Code segments were defined earlier as consisting primarily of instruction code, and being the basic entity for transfers of code between core and the swapping area.

As shown in figure 4-3, a code segment may contain several subprograms, each of which may have been separately compiled in the file area before becoming part of a segment in the swapping area. Subprograms may be further broken down into a mixture of procedures and serial code, but this is dependent on the nature of the source language and specific coding.

One important point to note about code segments is that, since code cannot be changed after it is compiled, the copy of a segment in the swapping area is identical with any copy that has been transferred into core. Thus when the operating system decides to swap out a code segment, no actual transfer needs to take place. The operating system simply makes note that the segment is now absent, and may then overlay the core area occupied by that segment. This is unlike the data segment which, being constantly subject to change, must be physically transferred to disc if swapping is required. (Note unidirectional arrows for code segment swapping and bidirectional arrows for data segment swapping in figure 4-3.)

CODE AND DATA SEGMENTS

The preceding introduction provided a bridge between the external aspects of the system and the inner workings of the hardware, which now follow. Attention is to be focused on main core memory, regarding the swapping area only as a place where segments can be sent when main memory becomes too crowded.

At first, memory will be viewed as a whole, as a repository for some number of segments — whether they be code or data — with perhaps some spaces between. It will be shown how space is managed in an orderly and efficient manner. Following this, code segments and their interrelationship during execution will be discussed, followed finally by data segments and the stack concept.

SEGMENTS IN MEMORY

Figure 4-4 shows three segments being present in core memory. (The actual number could be up to 255.) They are separated by three blank segments, such as might typically be the case during operation.

To assist the operating system in its task of filling core with variable sized segments, the memory is threaded with two major systems of links. These are the *main memory links* and the *free space links*. The main memory links contain pointers which link all segments, both used and free, while the free space links are responsible for linking only the free (or blank) segments. (Figure 4-4 is a simplified form of the link structure used by the operating system, and should not be construed as documentary.)

Linking pointers are given for both the forward direction and the backward direction. Note first the main memory links. These consist of a few words preceding every used and free segment. The first word is the segment head, and it includes information to state which type the segment is — used or free. A second word contains an absolute address pointing to the segment head of the next segment in core. This is the forward direction (i.e., to a higher address). Note that the forward pointer for the last segment wraps around the end of core to point at the first segment. A third word contains an absolute address pointing to the segment head of the preceding segment in core — i.e., the backward direction. Wraparound also occurs in this case, from the first segment to the last segment.

The free space links are similar to the main memory links, but are embedded in the segment rather than preceding it. The first word contains an absolute address pointing to the third word of the next free segment. Similarly, the second word points back to the third word of the preceding free segment. The significance of pointing to the third word is that it contains the size of the free segment. This makes the sizes of free segments easy to reference; in fact the operating system may easily find segment space of a desired size by issuing only one instruction, the Linked List Search (LLSH) instruction.

When segments are deallocated or overlaid, the links are automatically updated by the operating system. For example, if the middle filled segment shown in figure 4-4 should become deallocated, it will, together with the blank segments immediately preceding and following that segment, form one larger blank segment. The intervening links are eliminated, and the links preceding and following this new larger block are extended to define one contiguous blank segment.

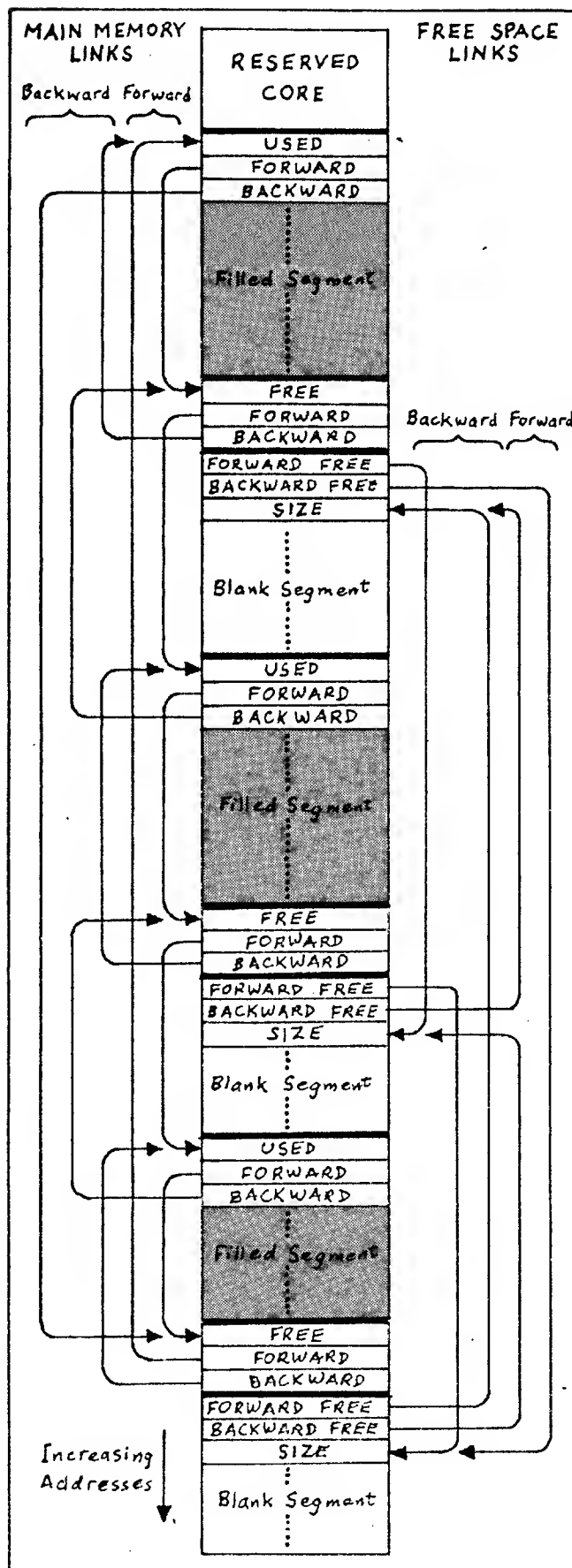


Figure 4-4. Contents of Core Memory

As another example, suppose that a segment coming in is too large to fit any of the blank spaces, and the operating system determines (again) that the middle segment must go. The operating system swaps the middle segment out to the disc, and begins allocating the new segment immediately following the top segment. At some point the allocation ends, leaving a certain amount of blank space from that point to the next filled segment. The operating system accordingly establishes all the necessary new links to restore an accurate portrayal of memory space.

CODE SEGMENTS

During the execution of one user's process, there will typically be several code segments in core and a single data segment. Assume that the current process presently has two code segments in core, as shown in figure 4-5. (The data segment, not shown, will be discussed later.)

The purpose of figure 4-5 is to show how the system keeps track of where code segments are, and how references may be made from one segment to another. Although the figure illustrates hardware, it remains the responsibility of the operating system to control the actions shown here.

The Code Segment Table and the CST Pointer have both been mentioned before. In summary, it was explained that the CST Pointer is permanently resident in location 0, and that it contains an absolute address pointing (1) to the starting location of the Code Segment Table. This table tells where each code segment (present or absent) is located.

Each entry in the Code Segment Table has a unique number, called the *code segment number*, which identifies a particular segment. Each entry consists of a doubleword descriptor which includes the absolute address of the related segment and its length. (The format of CST entries is given in figure 4-6.) Entry number 0 in the table is unique in that it simply points (2) to the final entry in the table; this defines the length of the table for the benefit of the operating system in allocating core space for the table itself. Segment number 0 does not exist.

The example Code Segment Table in figure 4-5 presumably has 212 entries for all code segments of all users currently on the machine. Assume that one user is executing a process which requires code segments 22 through 25. Segments 22 and 23 are required to be in core, since there is a reference that has caused a link between them, whereas segments 24 and 25 are not presently needed and so are absent on disc.

The process is currently executing instructions in segment 23. This means that the address value contained in the second word of CST entry 23 has been loaded into the PB-register. Thus the PB-register is pointing (3) at PB(a). The PL-register, using a value derived from the segment length, is pointing at PL(a). The P-register is advancing from PB(a) toward PL(a), and instruction referencing is relative to the current values of the P-register.

The last nine locations of segment 23 are not part of the coded segment, but were added by the operating system when the segment was loaded into the virtual memory. This is the *Segment Transfer Table*, which contains linking references for every *procedure call* in the segment. A procedure call is an instruction which references a set of instructions elsewhere in the code segment; that set of instructions is structured as a *procedure*, to perform a standardized operation or computation and then return control to the instruction immediately succeeding the call instruction.

Note that entries in the Segment Transfer Table are numbered from the end back towards the code. Entry number 0 gives the Segment Transfer Table length (see STT Length word format in figure 4-6). This indicates (4) the number of the last STT entry, so that the hardware can make validity checks on procedure call references; for example a call to entry number 9 would be invalid. (If a call to entry 0 is made, the reference will be taken from the top of the stack instead of from the Segment Transfer Table.)

When the execution sequence reaches the first PCAL instruction, a call is made (5) to the Segment Transfer Table. The call requests the fourth entry in the table; i.e., since the PCAL instruction uses PL- addressing, the instruction references cell PL-. This location contains a *local program label* (see format in figure 4-6), which implies that the called procedure is located within the same segment. The reference is a PB relative address pointing (6) to the beginning of a procedure or block.

After some preparatory operations, which include saving the return address on the stack, the PCAL instruction transfers control to the procedure. Upon encountering an EXIT instruction in the procedure, control returns to the instruction immediately following the first PCAL.

In this example there were no references outside the current segment. In the following example an external reference is made.

When the execution sequence reaches the second PCAL, another call is made (7) to the Segment Transfer Table. The call requests the fifth entry in the table, which happens to be an *external program label*, indicated by a "1" in bit 0 (see format in figure 4-6). This implies that the called procedure or block is in some other segment. The contents of the label tells which segment, and also gives the STT number in that segment which contains the local reference.

The PCAL instruction, after the usual preparatory operations, transfers control to the called procedure as follows. The segment number given in the external program label points (8) to a specific entry in the Code Segment Table; this is assumed to be entry number 22. A value for PB is picked up in the second word of this entry, and is loaded into the PB-register. This causes the PB-register to point (9) to the starting location of code segment 22 (PB(b)). The limit (PL(b)) is also established. Meanwhile, the STT value given in the external program label is pointing (10) to entry number 4 of the Segment Transfer Table. This causes a PB relative address to be picked up for

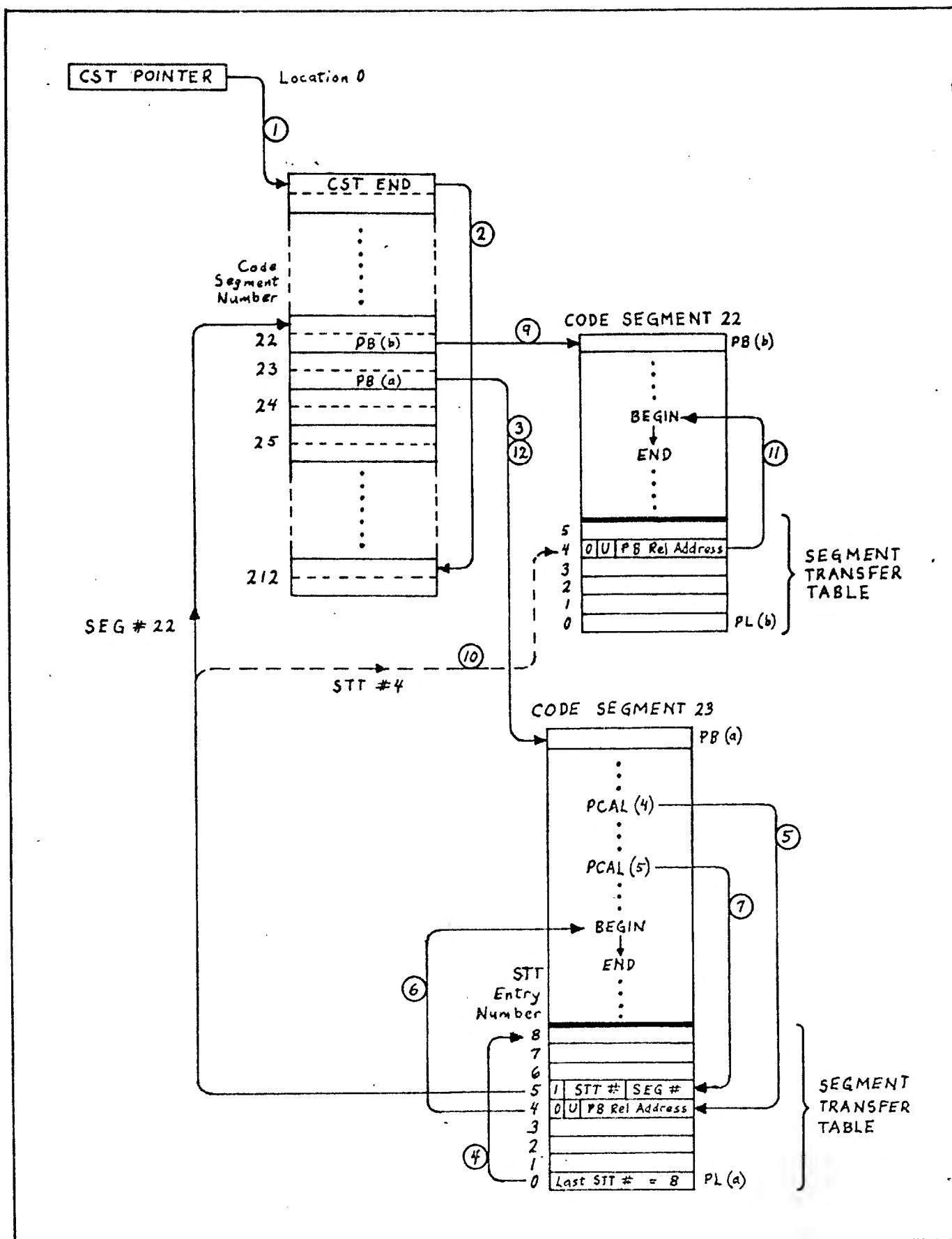


Figure 4-5. Procedure Calls Within and Between Code Segments

CODE SEGMENT TABLE

Doubleword

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	M	T	R	LENGTH											
ADDRESS															

A

Absence bit (=1 if segment is absent)

M

Mode bit (=1 if privileged mode)

T

Trace bit (=1 to call Trace routine)

R

Reference bit (for statistical use by operating system, set to 1 when accessed)

LENGTH

This value times 4 (max = 16,384)

ADDRESS

Absolute memory address (for PB) or absolute disc address if absent

SEGMENT TRANSFER TABLE

Words

STT Length

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	U	0	0	0	0	0	0	LENGTH							

U

Uncallable bit

LENGTH

Maximum = 255 (Calls from external segments may reference only the first 128 entries, PL thru PL-127.)

Local Program Label

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	U	ADDRESS													

U

Uncallable bit

ADDRESS

PB relative, + only

External Program Label

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	STT #								SEG #						

STT #

STT entry number in target segment, maximum = 127

SEG #

Target segment

STATUS

Word

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	I	T	R	O	C	CC	SEGMENT #								

M

Mode bit (=1 for privileged mode)

I

Interrupt enable (1)/disable (0), external

T

Traps enable (1)/disable (0), user

R

Right Stack Opcode bit (pending = 1)

O

Overflow bit

C

Carry bit

CC

Condition Code

SEGMENT #

of the caller

Figure 4-6. Formats Associated with Code Segments

the P-register. The P-register now points (11) to the starting address of the procedure or block, and execution begins. (If an STT number of 0 is given, execution would start at PB(b).)

Calling procedures outside of the segment in this manner is subject to a number of rules, checks, and safeguards. These ensure that the call is allowable, and that other users are fully protected from deliberate or accidental invasions of privacy. The way in which the operating system sets up the Segment Transfer Tables ensures that all transfers are legal for that process. Even if the user transfers via the top-of-stack reference into another user's code segment (assuming that it is callable) he can do no worse than execute part of that other segment. He will certainly render his own stack data meaningless, and furthermore can in no way read or relocate the other user's code or data. His end result is completely unpredictable, but would likely eventually invoke one of many possible error traps.

In addition, if the operating system ascertains that a local reference in a segment is of a category that will not normally have external references to it, the operating system will set the *uncallable bit* in the STT entry. When this bit is set, no external references in user mode may be made to that procedure or block. One typical application of this bit is to prohibit direct user access to the uncallable intrinsics of the operating system — i.e., those operations that the operating system will perform on behalf of a user, but cannot be directly accessed by the user.

At the conclusion of the called procedure, control is returned to the original segment by the EXIT instruction. This instruction looks at the Status register, which saves the segment number of the caller (see format in figure 4-6), and thus (12) returns the PB-register value back to PB(a). The saved P relative address on the stack re-establishes the return point, and execution continues at the location immediately following the second PCAL instruction.

DATA SEGMENTS

In the introductory paragraph under "Code Segments" it was stated that one user's process may have several code segments, but only one data segment. The following few pages deal with the data segment, particularly concentrating on the stack area of that segment.

As a beginning point of reference, figure 4-7 shows how the operating system establishes and keeps track of a particular data segment. As indicated by a note in the figure, this is accomplished by tables maintained by — and known only to — the operating system.

Assuming we are working with processor number 1 of a single- or dual-processor system, core location 4 contains the *Current Process Control Block pointer*. In the example shown, this pointer (1) has selected process number 31 by pointing to that particular block in the Process Control Block table. This means that process number 31 is *currently* being executed on the machine.

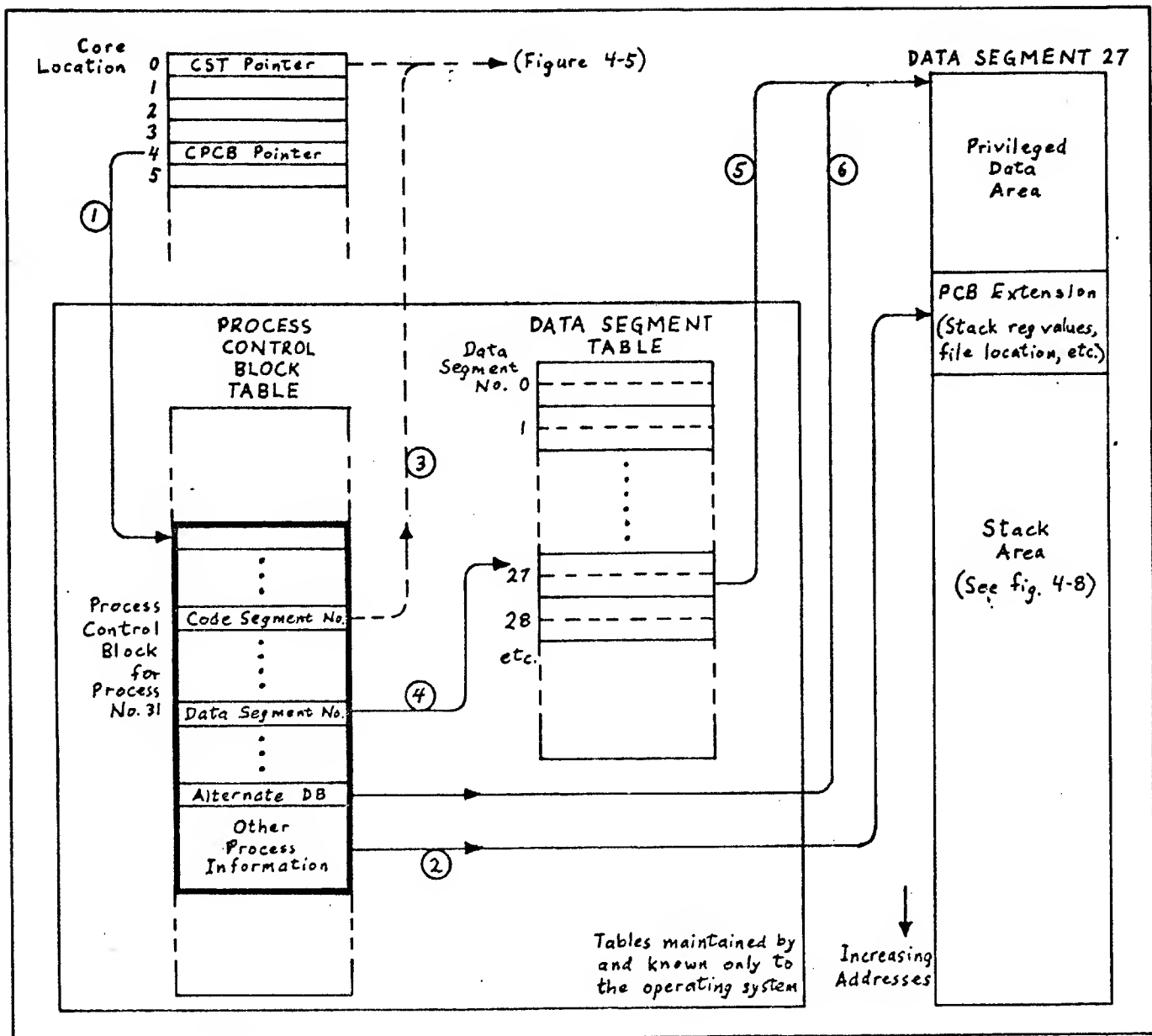


Figure 4-7. Locating the Stack for One Process

The Process Control Block contains considerable information pertaining to the control of that process, such as priority, queue pointers, descendency pointers, and so on. Included in this information (2) are saved stack register values and other information which is actually contained within the segment. This area of the segment is the *Process Control Block Extension*.

However, relevant to the present discussion, the most significant information consists of the code segment number(s) and the data segment number. The absolute address contained in the CST pointer location, indexed (3) by the code segment number, identifies a particular code segment as shown earlier in figure 4-5. The data segment number points (4) to a doubleword descriptor in the Data Segment Table.

Assuming that the data segment for this process is number 27, entry number 27 in the Data Segment Table will be pointed to. The second word of this entry will give an absolute address pointing (5) to the beginning location of the segment.

The data segment itself includes three separate areas, one of which is the PCB Extension already mentioned. The *privileged data area* is an auxiliary data area (i.e., additional to the stack area), which is available to the user by means of an alternate data base value (6) provided by the Process Control Block. When this area is accessed, the DB-register takes the alternate DB value, and addressing can then be positive with respect to this value. Figure 4-7 shows this area to be contiguous with the rest of the segment (thus alternate DB also points to the beginning of the segment),

but this may not necessarily be the case. The operating system may place the privileged data area in some other part of core.

The remaining portion of the data segment consists of the stack area. The stack is where most dynamic computational operations take place, and it is the next major subject of discussion. The study of the stack, its operation and effects, will occupy the remaining portion of this section.

The *stack* can be defined as a linear list of data in which the last element added to the list is in the prime position for computational operations (comparable to an accumulator), and is the first element to be removed when the program needs data from the stack. This type of data structure is also more strictly identified as a "LIFO" (last in, first out) stack, since data is removed from the stack in the reverse order from which it was added.

Although many instructions can reference elements within the stack, it is the element currently on the top of the stack which is of greatest significance. Note that the top element of the stack will be a different word, occupying a different physical location, each time data is added to or deleted from the stack. However, that top element has an identity, to both hardware and software, and is termed the *top-of-stack* element. It is also known by its acronym, TOS, and loosely as the top of the stack.

Figure 4-8 shows the basic construction of the stack area and the way stack registers in the CPU delimit the various parts. Remember that there will normally be several stacks in memory, one for each process, but only one will be active at a given time. The stack registers point to the currently active stack.

The stack area is bounded at the low end by the DL-register and at the high end by the Z-register. A major division into two parts is delimited by the DB-register, which points to the base location of the stack. The area between the DB and DL locations is not part of the stack itself, but is closely associated with the stack by providing an area for dynamic own arrays. Except for programmatical applications, this area is not particularly significant, and will be ignored in the following discussions. Its existence, however, should be acknowledged.

Just as the DB-register points to the base location of the stack, so the SM-register points to the current top-of-stack location (in memory). The convention of drawing stack diagrams corresponds to the manner in which code is written (or any written language), beginning at the top of the page and proceeding to the bottom. Thus the stack appears inverted, with the last entry (top-of-stack) toward the bottom of the diagram. Addresses increase in a downward direction.

Whereas the DB-register and Z-register contents are static, the SM-register content is constantly changing as the program progresses, moving up and down the stack area. At all times, the area between DB and SM is filled with valid data, while the area between SM and Z is available for additional

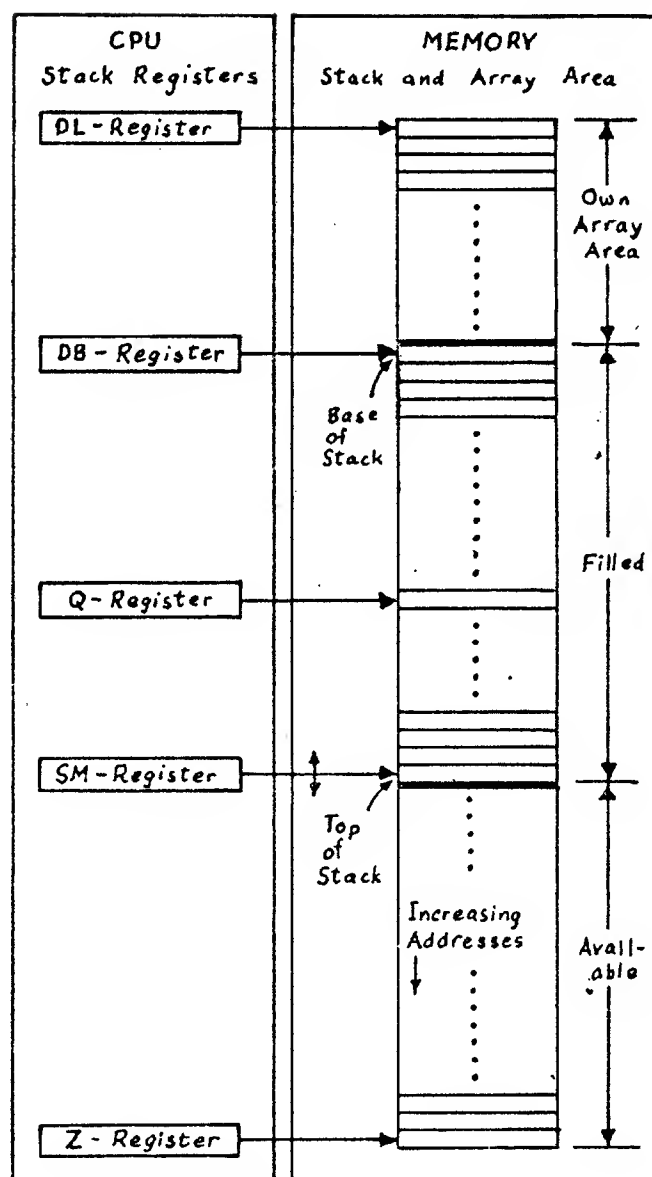


Figure 4-8. Stack Registers and One Stack

data. Should the quantity of data exceed the available space, the attempt to move SM past Z will invoke an interrupt to the operating system, which may grant additional space (new Z value), one or more times—within certain limits.

Unlike the fluid cell-at-a-time movement of the SM pointer, the Q-register value moves sporadically in jumps. It is the purpose of the Q-register to retain the starting point of data relating to the current procedure. Thus when a new procedure begins, the Q pointer jumps ahead to establish a new starting point at the current top of the stack. Conversely, when a procedure ends, the Q pointer jumps back to the place it had marked earlier for the preceding procedure. This action will be illustrated shortly.

As far as the current procedure is concerned, its stack data consists of the locations from a "base" of Q to the current top of the stack.

In the foregoing discussion of basic stack structure, the SM-register was assumed to point at the absolute top of the stack. This is true only for the portion of the stack "in memory". In actual fact, provision is made to allow a few top words of the stack (maximum of four) to "spill over" into hardware registers in the CPU. This is shown in figure 4-9, where the three topmost words are actually in the CPU. The SM-register points to the last stack element in memory, but the actual top-of-stack is in the third CPU register. The actual top of the stack is designated as S.

The four registers in the CPU reserved for receiving top stack elements are scratch pad registers employed only by the CPU hardware. They may not be addressed externally. Externally, the programmer is interested only in the S location contents. The hardware defines the address for him to be at (in this example) the SM-register value plus 3. The value 3 is retained in the SR-register, a three-bit register, which will never indicate a value higher than 4.

The address value S obtained by adding the SR-register contents to the SM-register contents is a completely valid address. In fact, when the CPU registers must be cleared for some other operation (e.g., a new procedure or an interrupt), the register contents are physically transferred to the numerically corresponding memory locations. In this example, the SM pointer would move up by three locations, and the SR-register content would become 0.

Again it must be stressed that the user is not aware of these registers. The reason for their existence is speed. For example, it is possible to perform computations on the four top elements of the stack without making a single memory fetch.

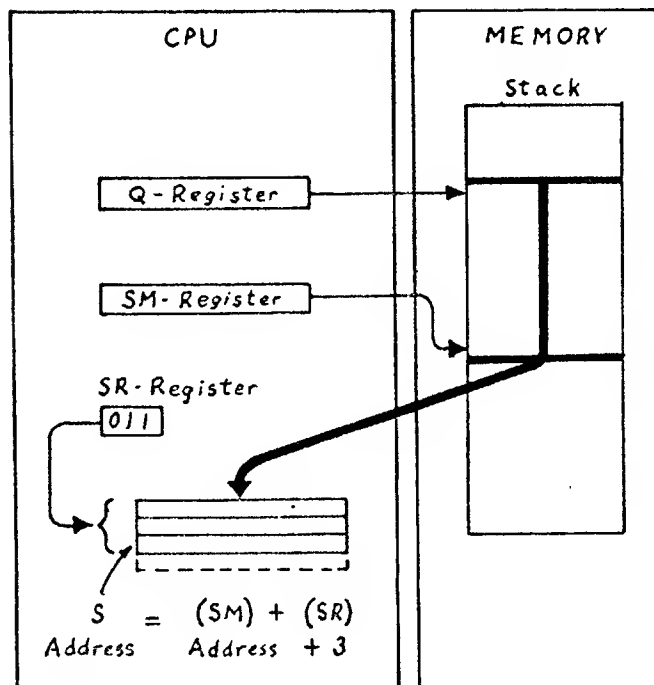


Figure 4-9. Top-of-Stack in the CPU

Since the actual top of the stack (S) is the value of interest, and since S is a valid address, the separate existence of SM and SR values is commonly disregarded, as in the following discussions.

The action of the Q-register in marking the starting location for each procedure's data is shown in figure 4-10.

This figure will be discussed in detail, but briefly, what has occurred in the example shown is the following. The currently executing code segment was working with data in the temporary storage area immediately following the "first Q" location. At that time, the Q-register was pointing at "first Q", S was indicating the top of the stack, and the Z-register was pointing to the end of the data segment. If the executing code segment never called a procedure, the stack picture would never get more complicated. However, at some point the code called a procedure (perhaps a lengthy mathematical routine) by means of a PCAL instruction. This caused additions to the stack as indicated (procedure A). New data was incurred as the procedure began, and S pointed to the top of that data as it was generated. Then procedure A called procedure B (perhaps a frequently used equation), which resulted in new additions to the stack, as shown. Then still later, procedure B called procedure C (perhaps a library routine for a trigonometrical function), resulting in a final picture of the stack as shown.

What will happen next is that procedure C will end, saving its answer in a convenient place for procedure B to access, and issuing an EXIT instruction. Then all the other stack additions due to procedure C will be eliminated (by moving the S and Q pointers back), and procedure B will continue its computations on its own stack data. Likewise, procedure B will come to an end, save its data, and exit, resulting in the elimination of the procedure B stack data. And finally procedure A will do the same, returning the net answer to the new top of the stack, on the main temporary storage area.

It is obvious from this brief outline of events that each time control is returned from the called procedure to the caller's procedure — within the code segment — the stack registers also return to the caller's data area. Thus the stack mark chain virtually eliminates system overhead in keeping track of lexicographical levels (nesting of procedures). For example, the simple return sequence described above, C-to-B-to-A-to main program, is not imperative. Procedure C could have been called again before the return to the main program was complete. Or other procedures (D,E,F, etc.) could enter the picture. But the return for both code and data will always remain perfectly in step — from the called to the caller.

Now the details. Beginning at the top of figure 4-10, note that the area between DB and the first Q is the *global data area*. The locations in this area are reserved by the process for variables (possibly arrays) which it has declared to be global for all procedures incurred by that process. That is, any procedure using this particular data segment may reference the variables in this area.

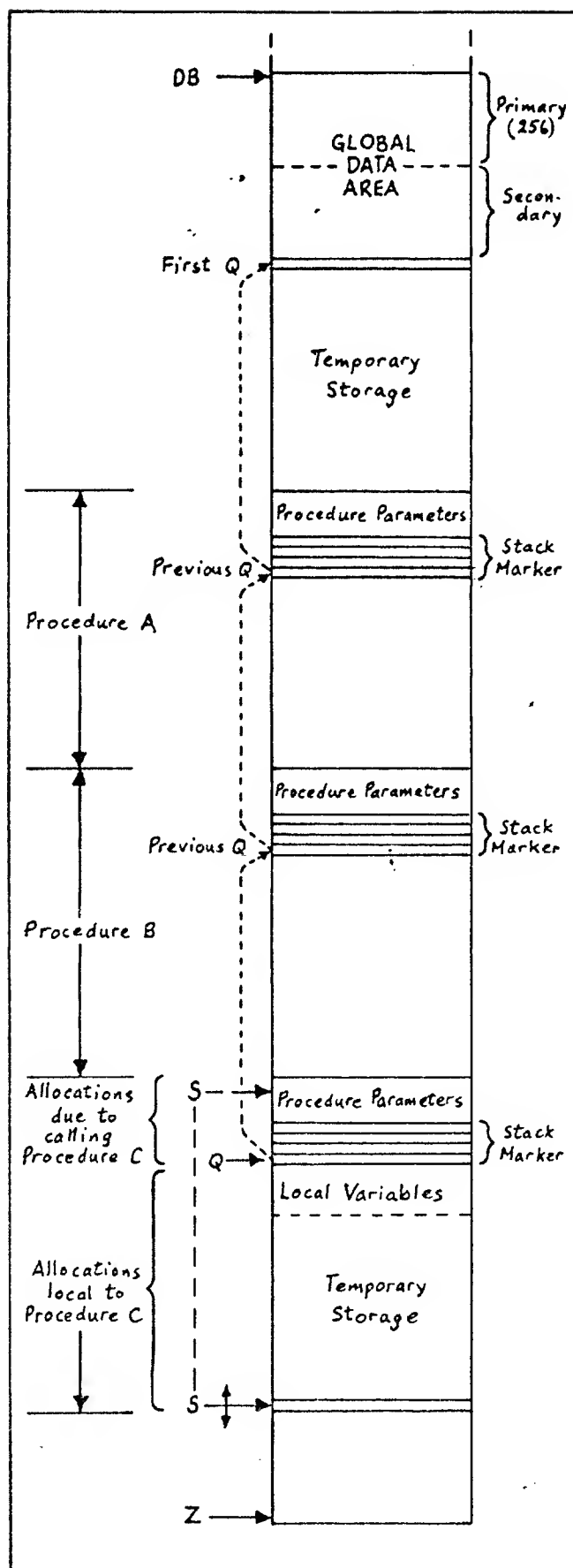


Figure 4-10. Stack Mark Chain

The individual locations in the global data area may contain an actual value, or may contain an indirect address pointing to some other location. (That other location either will contain the value or will be the start of an array.) Since DB relative addressing is limited to a maximum of DB+255, only the first 256 locations of this area may be addressed directly. These locations are denoted as the primary global data area. If the number of entries exceeds 256, indirect addressing must be used. Locations in this area (convenient for arrays) are denoted as the secondary global data area.

When the operating system finishes assigning space for the global variables, it points the Q-register at the next succeeding location (first Q). This is the actual start of the stack proper. Initially the S pointer is also pointed at this location, since there is as yet no data on the stack. As the executing code segment proceeds to obtain, manipulate, and generate data for the stack, the S pointer moves away from Q, indicating at all times the top of such data. (Examples of typical operations will be given under the next major heading, "Examples of Stack Operation".)

Then at some time during execution of the code segment, it is assumed that Procedure A is called. Accompanying the call are a set of *procedure parameters* which are placed on the stack just prior to issuance of the PCAL instruction. These are actual parameters, to be substituted for formal parameters in the procedure, and are referenced by Q-addressing.

Calling the procedure causes a four-word *stack marker* to be placed on the stack. The format of this marker is shown in figure 4-11. The first word saves the current contents of the X-register. The second word saves the return address for the code segment—i.e., the P-register address (plus one) relative to the PB-register contents. The third word saves the Status register contents, which includes the code segment number of the caller, in case the called procedure is external to the current code segment. (This was described earlier under "Code Segments".) The fourth word is the one of most interest to the present discussion. This word contains the *delta Q* value, which tells how far back it is to the previous location to which Q was pointing. In this case, delta Q is pointing to "first Q". The Q-register now points at this delta Q location.

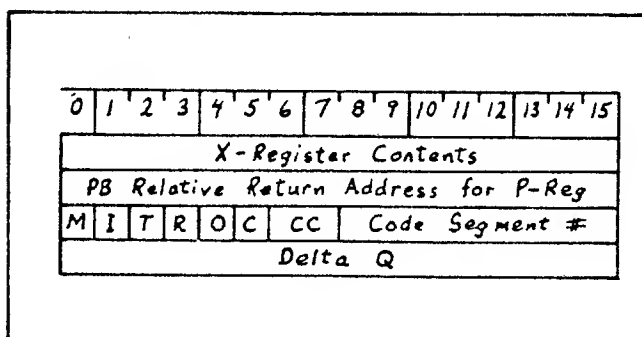


Figure 4-11. Stack Marker Format

The sequence of events described in the preceding two paragraphs is repeated when procedures B and C are called. Each time, the Q-register will point to the delta Q location of the current stack marker, and the contents of that location will point back to the previous setting of Q. Thus it is seen that when procedure C is executing, there will be a chain of delta Q stack marks linking the present Q setting back to the first Q.

Just as the links are established as the procedures are called, so are they used and eliminated as the procedures are exited. When procedure C ends, the EXIT instruction returns S to equal Q, essentially placing the delta Q value temporarily on the top of the stack. This allows the EXIT instruction to compute a new value for the Q-register ("previous Q"), and it appropriately moves Q back. The EXIT instruction causes S to decrement step-by-step through the stack marker, restoring Status, P-, and X-register contents for procedure B.

Lastly, S is moved back to eliminate the unwanted parameters of procedure C. Presumably one or more parameters will be computed answers resulting from procedure C, and so S is only moved back so far as to preserve those desired answers (which are now on the top of the stack). This ability to move S back selectively is one of the functions of the EXIT instruction (refer to instruction definition).

Once again, the sequence of events described in the preceding two paragraphs are repeated, until all procedure data and stack marks are eliminated, and only the final answer is on the top of the stack.

As a final note, observe the breakdown of allocations for one procedure (procedure C illustrated). As shown, the procedure parameters and stack marker are allocations due to calling the procedure. The remaining locations are allocations local to the procedure, which are further broken down into an area for *local variables* and an area for *temporary storage*.

EXAMPLES OF STACK OPERATION

Up to now, the mechanics of the stack have been examined without the application of specific values or problems. To conclude this section, various examples of stack operation will be given. The examples are progressively instructive and, in each case, the advantages of this type of architecture over the register structured computer will be illustrated.

The examples do not necessarily show all the advantages of a stack machine. In fact one of the major advantages has already been shown — that of preserving code and data conditions by marking the stack. This facilitates rapid environment changes (e.g., swapping users), saves overhead

for unlimited nesting of procedures, and helps to make code re-entrant. Another major advantage, that it allows fast interrupt handling, will be covered in a later section. The following examples are primarily designed to aid in understanding the stack concept.

BASIC ARITHMETIC

Figure 4-12 shows a sequence of basic instructions being executed on some data which is presumed to exist in the stack. The upper row shows the most elementary method of adding and removing data to and from the stack, via load and delete instructions. The lower row shows the effects of four arithmetic instructions.

As shown for the initial stack condition (A), the data consists of six numbers in consecutive locations. The A-register points to the oldest element on the stack, and S points to the element currently on the top of the stack. A Delete instruction (DEL), executed between A and B, causes the number 44 to be removed from the stack; this is accomplished by simply decrementing the S pointer by one. Then, between B and C, a LOAD instruction causes the number 37 to be loaded onto the stack; this is accomplished by storing the number 37 (from another memory location) into the location formerly occupied by the number 44, and then incrementing the S pointer by one.

Between C and D, an ADD instruction is executed. This instruction adds the two top elements of the stack together, deletes both from the stack, places the answer (100) on the top of the stack, and points S at the answer.

Note

As mentioned previously, up to four of the top stack elements may exist in CPU registers. Obviously, to execute the ADD instruction, at least the two top elements must exist in the CPU. To ensure that this is the case, the hardware checks the content of the SR-register. If the number contained therein is not at least 2, one or more memory fetches are made so that the instruction can be carried out.

Between D and E, a Multiply instruction (MPY) is executed. This instruction multiplies the two top elements of the stack together, deletes both from the stack, places the answer (700) on the top of the stack, and points S at the answer.

To subtract (SUB), the top element is subtracted from the next-to-top element. Thus the answer at F is the result of 500-700, or -200. (As before, only the answer remains after computation is performed.) Finally, at G, negation is performed. This simply reverses the sign of the number on the top of the stack; in binary form a two's complement operation is performed.

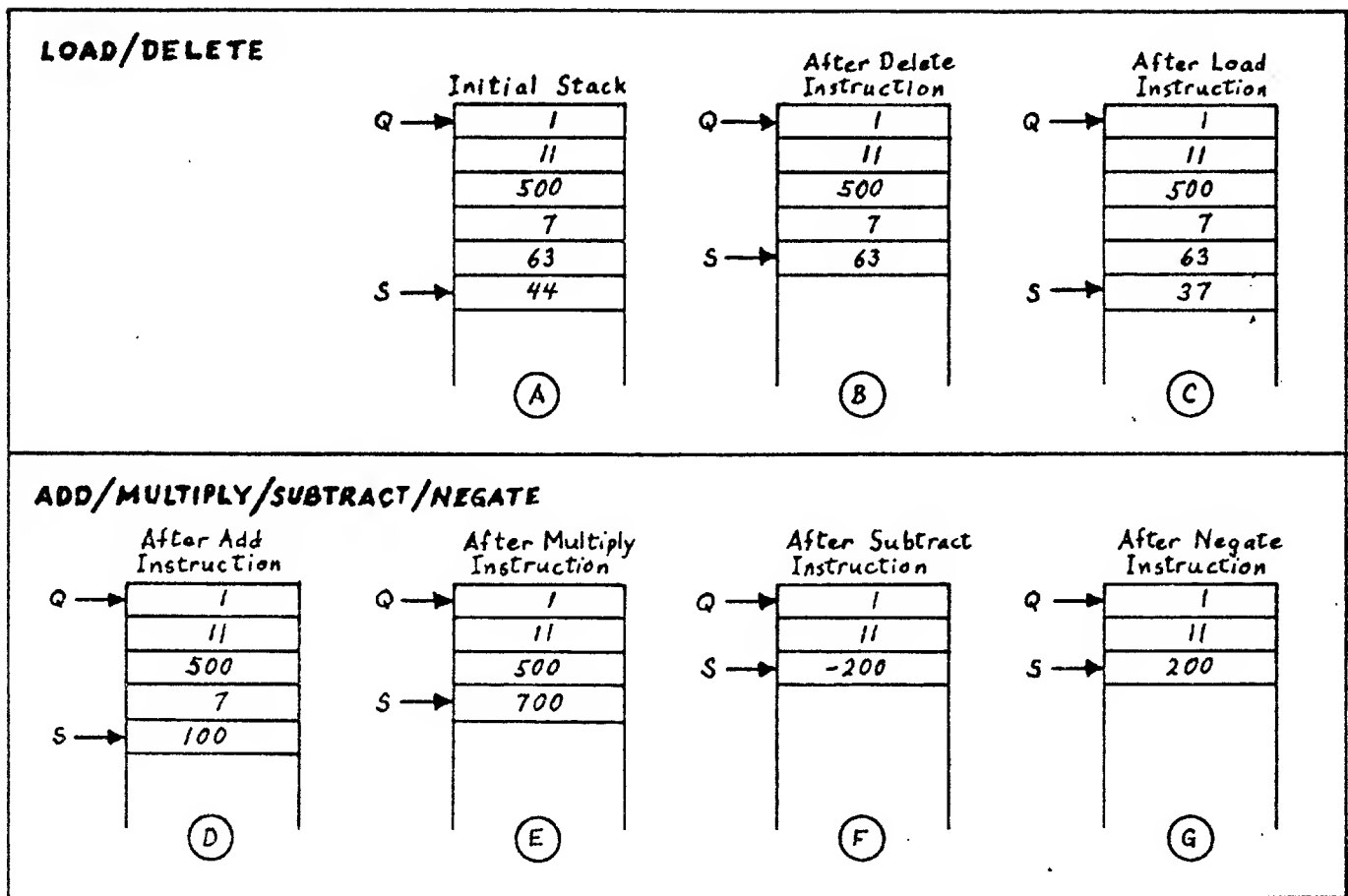


Figure 4-12. Basic Arithmetic Stack Operations

Although the sequence A through G in figure 4-12 is a very simple series of operations, it does illustrate the advantages of the stack technique in computation. First, note that regardless of how many elements of data there are or what memory cells they occupy, the operand for each instruction is consistently the same — the top of the stack. This permits *implicit addressing*; i.e., since the operand is understood to be the top of the stack, it is not necessary to give an operand address in the instruction word. Thus (except for LOAD, which must specify a relative address to load from), the instruction can simply say “add”, or “multiply”, etc. The immediate benefit of this is that it allows code compression. Two instructions can be given in a single word. The sequence D through G, for example, can be given in two instruction words. Since this reduces the number of memory fetches, the speed of computation is considerably increased.

A second point to note is that temporary storage of intermediate results is automatically provided. For example, once the parameters 63 and 37 (at C) have been added, they are no longer required and so are thrown away. But the answer, which is substituted on the top of the stack, is automatically in position (adjacent to 7) for the ensuing multiplication. Thus there is no need to provide a dedicated location to save the temporary quantity 100 (or any of the other intermediate results).

It is apparent that the order of placing elements on the stack is very important. However, it is one of the compiler's functions to provide the correct order, and (except in assembly mode) this is of little concern to the programmer.

PROCEDURE CALLS

Figures 4-13 and 4-14 illustrate the operations involved in a procedure call. Figure 4-13 shows programmatically how a procedure is set up and called, and figure 4-14 shows what happens to the stack when the procedure is called and executed.

The purpose of this example is to demonstrate the ease and simplicity of *parameter passing* — i.e., the means by which a program can substitute *actual parameters* for the *formal parameters* declared in a procedure. In this example (see bottom block in figure 4-13), the formal parameters are J and K, and the actual parameters to be passed to the procedure are 25 and 10, respectively.

As shown in the bottom block of figure 4-13, the calling of a procedure has an equivalency in mathematical terms. That is, a procedure is like a predetermined equation, in this case “ANSWER = J/K”. Calling the procedure is like a request to solve the equation for the specific values of 25 for J and

SOURCE LANGUAGE		
	1	BEGIN INTEGER ANSWER;
Pro- cedure	2	INTEGER PROCEDURE QUOTIENT(J,K);
	3	VALUE J,K;
	4	INTEGER J,K;
	5	BEGIN
	6	QUOTIENT ← J/K;
	7	END;
Call	8	ANSWER ← QUOTIENT(25,10);
	9	END;

MACHINE LANGUAGE		
	Assembly	Octal
Pro- cedure	10 LOAD Q-5	041605
	11 LOAD Q-4	041604
	12 DIV, DEL	002340
	13 STOR Q-6	051606
	14 EXIT, 2	031402
	15 ZERO, NOP	000600
Call	16 LDI, 31	021031
	17 LDI, 12	021012
	18 PCAL, 20	031020
	19 STOR DB+0	051000
	20 PCAL (to system)	031XXX

MATHEMATICAL LANGUAGE	
Procedure:	ANSWER = J/K
Call:	Solve ANSWER for J = 25 and K = 10
Execution:	ANSWER = 25/10 = 2, remainder 5
Note: Decimal 25 = Octal 31 Decimal 10 = Octal 12	

Figure 4-13. Declaring and Calling a Procedure

10 for K. Executing the procedure is to perform the computation, in this case getting an answer of 2. (To keep things simple, the example procedure will be made to work strictly with integer numbers; thus the fractional remainder 5/10 will automatically be discarded.)

The upper two boxes in figure 4-13 list two forms of the program that will accomplish the example procedure. The top box shows how the program would be written in the source programming language. The middle box shows the machine language code that would be emitted by the

compiler. The machine language code is shown both in assembly (or mnemonic) form, and in an octal form of the actual binary machine code.

Both the source and machine language versions of the program will now be considered on a line by line basis. First, the source language program.

Line 1 begins the program block, just as line 9 ends it. Although the entire program consists only of one procedure and a call to that procedure, it nevertheless remains necessary to enclose the program between a *BEGIN statement* and an *END statement*. These statements define a program. ANSWER is declared to be a *global variable* for this program by giving its name within the *BEGIN statement*. This will cause the variable ANSWER to reside in the global data area, and thus allow its access by another procedure — such as an output routine to print out the result. The type declaration *INTEGER* specifies that ANSWER will always be an integer, and tells the compiler to reserve one word for the result (rather than two or three).

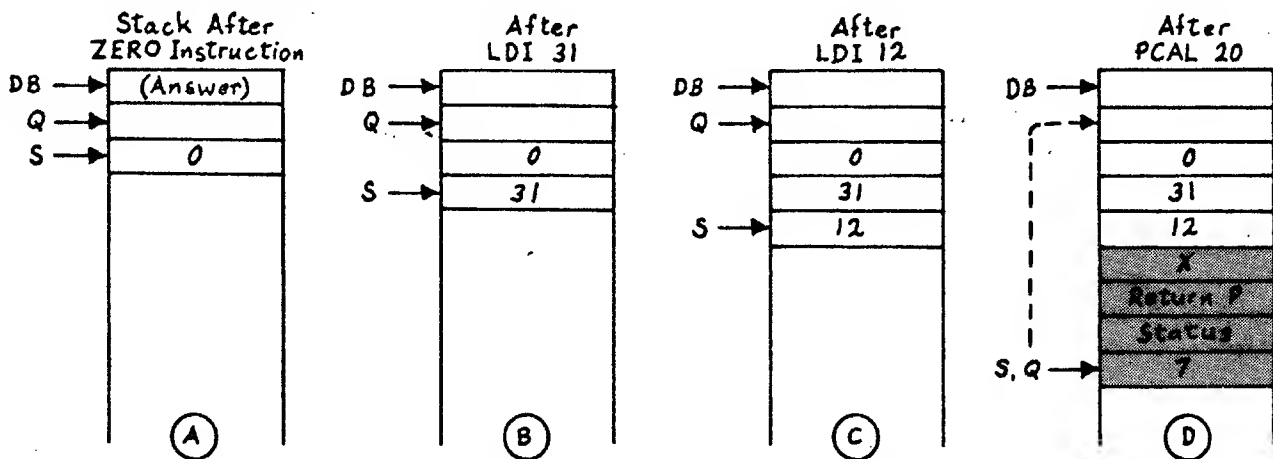
Lines 2 through 7 comprise the *procedure declaration*, which includes the *procedure head* (lines 2, 3, 4) and the *procedure body* (lines 5, 6, 7). The procedure declaration in a program cannot cause execution by itself; it must be called before any execution can take place. Thus the procedure declaration is always separate and distinct from the procedure call. They need not be immediately adjacent, as in this example.

Line 2 gives the *procedure name*, QUOTIENT, and declares that the procedure is of type *INTEGER*, which means that the result will be in integer form. It also gives the names of the formal parameters, J and K. Line 3 is the *value part* of the procedure declaration. Declaring J and K as values means that a value (rather than a pointer) will be passed as a procedure parameter, in both cases. This permits working with a copy and eliminates any need to change the actual parameter. Line 4 declares that actual parameters for J and K must be integers; if any other type is given (floating point, for example), a compilation error will result.

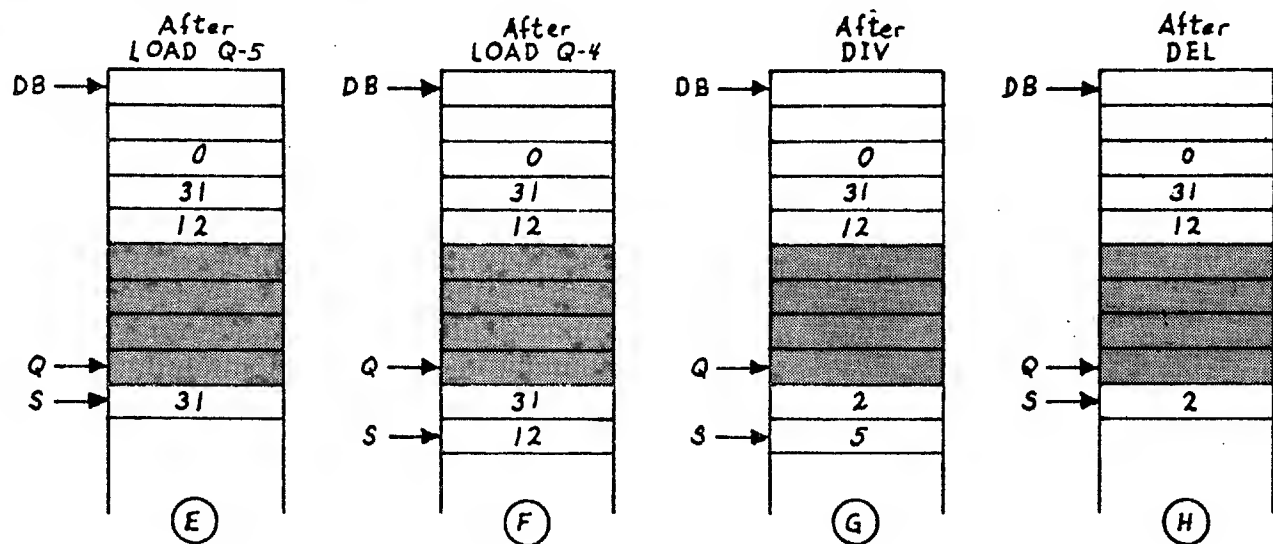
Line 5 begins the procedure body. Actually, since this procedure consists of only one statement, the *BEGIN statement* and *END statement* (line 7) are superfluous. They are included here, however, to illustrate the common form for a procedure (normally involving a compound statement). Line 6 is the *procedure statement*, the executable part of the procedure body. It is this statement which will cause the division of J by K, and will temporarily store the quotient as a procedure result, identified by the procedure name QUOTIENT.

The call to the procedure is given at line 8. This is an executable statement, as opposed to a procedure declaration. When this statement is encountered in a program, it will cause the procedure named QUOTIENT to be executed, passing actual parameters of 25 and 10 to the procedure, and will cause the global variable ANSWER to assume the value of the result. At this point (line 9) the program is complete.

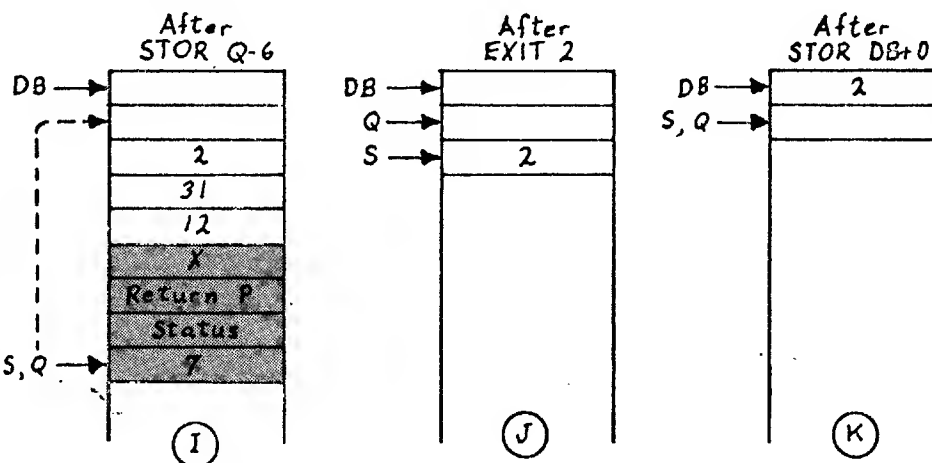
CALLING THE PROCEDURE



EXECUTING THE PROCEDURE



SAVING PROCEDURE RESULTS



Note: Gray shaded area is Stack Marker

Figure 4-14. Executing a Simple Procedure

Lines 10 through 19 show the machine language code which the compiler emits for the two executable statements in the program. That is, line 6 causes lines 10 through 14 to be generated, and line 8 causes lines 15 through 19 to be generated.

In order to explain the operation of the program in machine language, it is necessary to examine what is happening on the stack. Figure 4-14 will therefore be referred to in the following discussions. Furthermore, to aid in visualizing the operations, they will be described in chronological order; i.e., the machine language program will begin to execute at line 15.

First of all, it is assumed that the user has logged onto the system, has compiled the program, and is ready to run (or is running a program that will shortly encounter the statement in line 8). Loading the program has caused space to be allocated for the one global variable, ANSWER, which is at DB+0 (see A in figure 4-14). Since there are no other global variables, Q and S initially point at the immediately following location. (The content of that location will never be significant; in essence it is a dummy delta Q location.) It may be instructive to refer back to figures 4-10 and 4-8.

Additionally, during program loading, the operating system has evaluated the program in order to set the Z-register appropriately for an initial estimated stack size. Also, since no dynamic own arrays are declared, DL is set coincident with DB.

Now it is assumed that the user issues a system command to run the program or, in other words, to execute the procedure call given in line 8 of figure 4-13. This causes control to be passed to line 15 in the machine language program, where the sequence to call the procedure begins.

The first instruction is a ZERO, NOP. Executing this instruction puts a 0 on the stack and increments the S pointer (see A in figure 4-14). This reserves a location for the procedure result.

Next (B and C; lines 16 and 17), the parameter values 31 and 12 are passed directly from the instruction words to the stack (area reserved for procedure parameters). Octal notation is used for these values.

Then (D, and line 18) a procedure call instruction, PCAL, causes a four-word stack marker to be placed on the stack. The S and Q pointers point to the delta Q location of the marker, which now indicates 7 (the number of locations back to the initial Q location). It is assumed that entry number 20 in the Segment Transfer Table will direct the call to the correct procedure starting point.

Now execution of the procedure begins (line 10). The first two instructions (lines 10 and 11) load copies of the procedure parameters onto the top of the stack (E and F), using Q- relative addressing. The next instruction (line 12) divides the top-of-stack parameter into the next-to-top parameter, and substitutes the quotient (2) and the remainder (5) on the top of the stack, as shown at G. The

second half of the same instruction (DEL) discards the remainder word by decrementing S, as shown at H.

To save the result, the STOR Q-6 (line 13) first copies the top-of-stack into the location reserved for the procedure result, formerly occupied by a 0, as shown at I. Then it is possible to exit from the procedure. The EXIT instruction (line 14) restores Q to its initial setting, and the "2" included with the instruction causes S to move back two locations past the stack marker. As shown at J, this leaves the result, 2, in the location reserved for QUOTIENT — now on the top of the stack. The EXIT instruction also returns program control to line 19, which causes the content for QUOTIENT to be stored in the location for ANSWER in the global data area. This produces the final result shown at K.

Finally (line 20), a procedure call to the system returns control back to the system.

RECURSION

The last example in this series demonstrates the stack principles involved in a *recursive procedure*. A recursive procedure is one which calls itself one or more times during execution.

Recursion is a powerful programming technique which derives from the re-entrant capability of the code. The advantages and other considerations of this technique are beyond the scope of this manual, and the example to be given does not necessarily illustrate the niceties of the technique. Rather the example is intended to show only how recursion is accomplished on the stack.

The example chosen is purposely kept simple in order to provide continuity with the preceding example. (Note that the form of the source language program for this example, in table 4-2, is nearly identical to that of the preceding example in figure 4-13.) The procedure simply computes $N!$ (N factorial), where N is the formal parameter. The procedure will be called with an actual parameter of 4, so that computation of $4!$ will be: $1 \times 2 \times 3 \times 4 = 24$.

In essence, this problem consists of repetitively multiplying the previous product by a parameter which is incremented by one on each repetition. To provide a starting point (initial "previous product"), the value 1 is automatically given. The procedure is designed to perform this multiplication sequence by repetitively calling itself, after it has been called once by the main program. Thus for any N , the procedure will be called $N+1$ times. In this example there will be one call by the main program and four recursive calls.

Table 4-2 lists the source and machine language forms of a program block to solve this problem. Since the source language program is so similar to the preceding example, it need not be discussed at this point. The machine language

Table 4-2. Recursive Program

SOURCE LANGUAGE			
<pre> : : BEGIN INTEGER Y; INTEGER PROCEDURE FACTORIAL(N); VALUE N; INTEGER N; FACTORIAL ← IF N=0 THEN 1 ELSE N * FACTORIAL (N-1); Y ← FACTORIAL(N) END; : : </pre>			
MACHINE LANGUAGE			
PB Relative Addresses	Instructions	Octal Code	Comments
00114	LOAD Q- 004	041604	Load parameter
00115	CMPI, 000	022000	Test it for zero
00116	BNE P+ 003	141503	If not zero, branch to 00121
00117	LDI, 001	021001	If zero, load 1 as initial multiplicand
00120	BR 006	140006	Branch to 00126 (to Exit loops)
00121	ZERO, NOP	000600	Save space for intermediate product
00122	LOAD Q- 004	041604	Load parameter
00123	SUBI, 001	023001	Decrement for use as new parameter
00124	PCAL, 026	031026	Recursive call
00125	MPYM Q- 004	111604	Multiply parameter by TOS
00126	STOR Q- 005	051605	Store this recursion's product
00127	EXIT, 001	031401	Save the product and exit
00130	ZERO, NOP	000600	Save space for final product
00131	LDI, 004	021004	Load initial actual parameter
00132	PCAL, 026	031026	Main program's call to the procedure
00133	STOR DB 015	051015	Save final product in global area
00134	PCAL, XXX	031XXX	Return to system

form has been slightly changed to more closely resemble an actual program listing. Some assumed PB relative addresses are given for each instruction, beginning at address 00114. The assumption here is that this program block is embedded in a larger "main" program. (Note that the assigned STT entry for this procedure is assumed to be 026, and the global assignment for Y is DB+15.) The starting point for execution is at address 00130.

Figure 4-15 illustrates the program in flowchart form. Box 1 in the diagram calls the procedure (boxes 2 through 9), box 10 saves the result, and then control reverts to the main program at box 11. The procedure consists of two phases. The call phase begins when the procedure is called by the program, and is repeated four times. Briefly, what happens in this phase is that a succession of N values are placed on the stack, along with a space for intermediate

answers. The N values are decremented to zero and then the exit phase begins. This phase successively multiplies an accumulating product by each of the N values loaded on the stack in the call phase — in the reverse order. On each loop unneeded stack information is deleted, saving only the answer for that loop, until only the final answer is left. At that time (box 9) the final EXIT instruction finds that its return address points back to the calling block, and so the final answer is stored in the global area and control reverts to the main program.

As will be shown in the following detailed discussion, the return address check at box 9 is not literally a test for a specific address. Rather it specifies a return to the address given in each stack marker. Obviously the last return (first one placed on the stack) will be a return to the outer block.

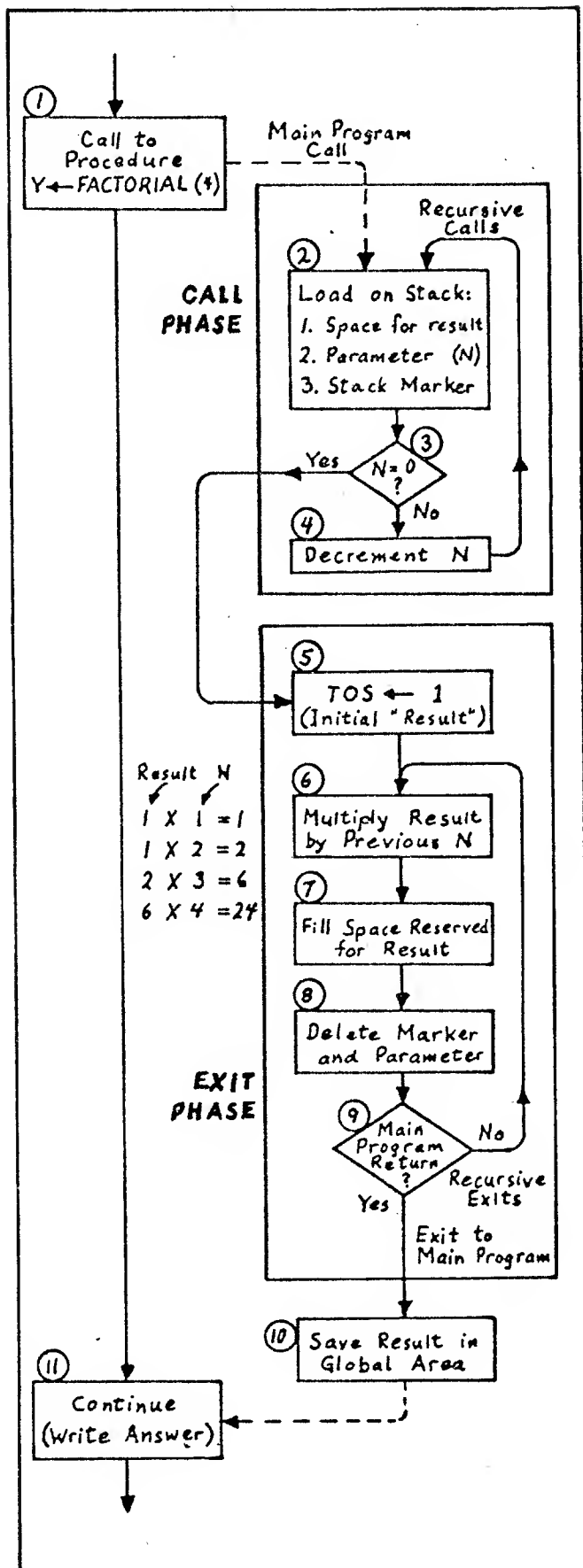


Figure 4-15. Example of Recursive Procedure

Figures 4-16 and 4-17 show the overall process of building up the stack by recursive calls, and then paring it down with recursive exits. These two figures will be used in the following discussions. Also the machine language program in table 4-2 will be referred to; individual lines will be identified by PB relative address, omitting the leading zeros.

MAIN PROGRAM CALL. As before, the main program has already reserved global space for the final answer (Y) before the procedure is called. When the call is given, the ZERO, NOP instruction at address 130 reserves space for the procedure result, FACTORIAL. (Compare stack pictures A and Z.) This is the first stack addition due to calling the procedure.

Next, the actual parameter 4 is loaded on (B), and then the PCAL instruction is issued. This causes the first stack marker to be loaded (C). This marker differs from the ones which will follow in that it contains return information to the outer block which called the present procedure. That is, the "return P" word is a P relative address for return to the caller in the code segment, and delta Q points back to the Q value that the caller was using earlier in the stack. Now, S and Q are both pointing at the last word of the first marker for this procedure.

TEST FOR ZERO. At addresses 114 and 115 (stack pictures D and E), the procedure parameter is first tested for zero. This is done by copying it onto the top of the stack (LOAD Q-4) and giving a CMPI 0 instruction. This instruction sets the condition code according to comparison results and deletes the tested word (E). Since the first test is non-zero (i.e., 4), the branch instruction at line 116 transfers control to address 121 (i.e., P+4). This test and branch will be repeated in each of the following recursion loops until the parameter has become zero.

FIRST RECURSIVE CALL. The branch to address 121 causes the procedure to call itself. As usual, the first action of the call is to load the procedure parameters onto the stack. The parameters in this case are the variable FACTORIAL and a decremented form of the original passed parameter. Thus the ZERO, NOP instruction reserves a location for FACTORIAL (see F), strictly for use by this recursion (i.e., distinct from the final FACTORIAL location reserved at A); then (G,H) the new parameter is obtained by copying the preceding value to the top of the stack (LOAD Q-4) and decrementing with a SUBI 1 instruction.

After loading parameters for the new call, another PCAL instruction is issued. This causes a new stack marker (see I) and, via the Segment Transfer Table, transfers control back to the starting point of the procedure, address 114. The new stack marker gives as its return P value the address immediately following the PCAL, which is 125. (This will be important to remember when the exit sequence is discussed.) Also, the delta Q value is 6, since the previous delta Q was six locations back.

SUCCESSIVE RECURSIONS. Now all of the steps described in the preceding three paragraphs are repeated,

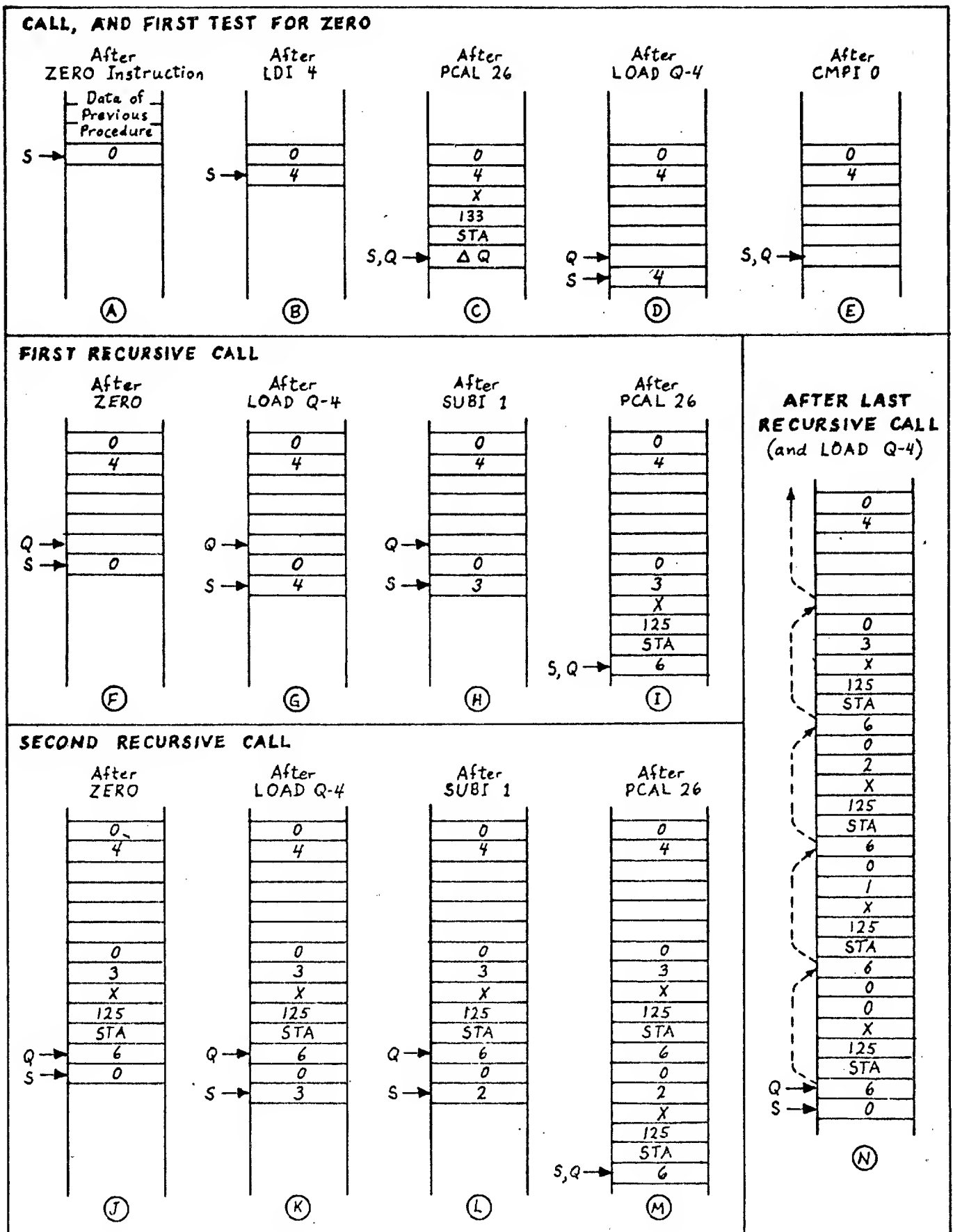
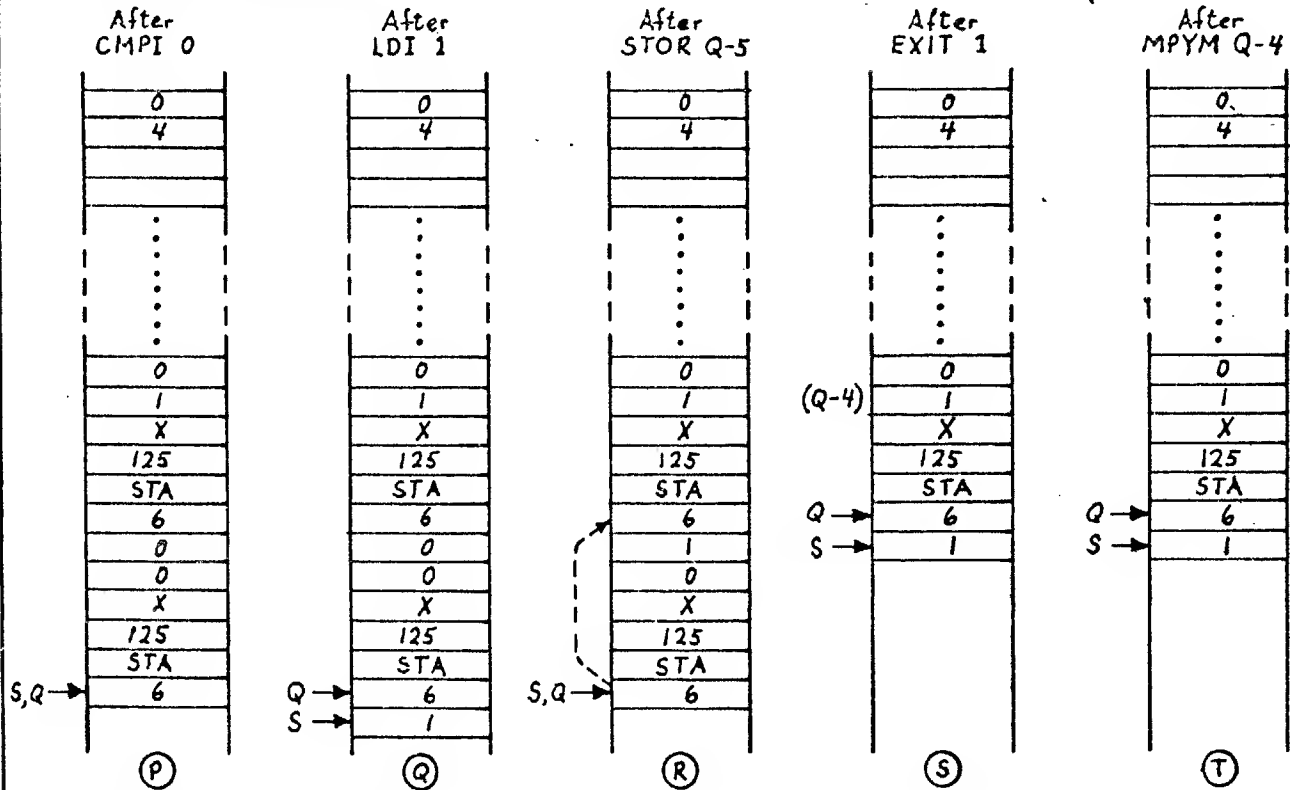


Figure 4-16. Recursive Calls

FIRST MULTIPLICATION



SECOND MULTIPLICATION

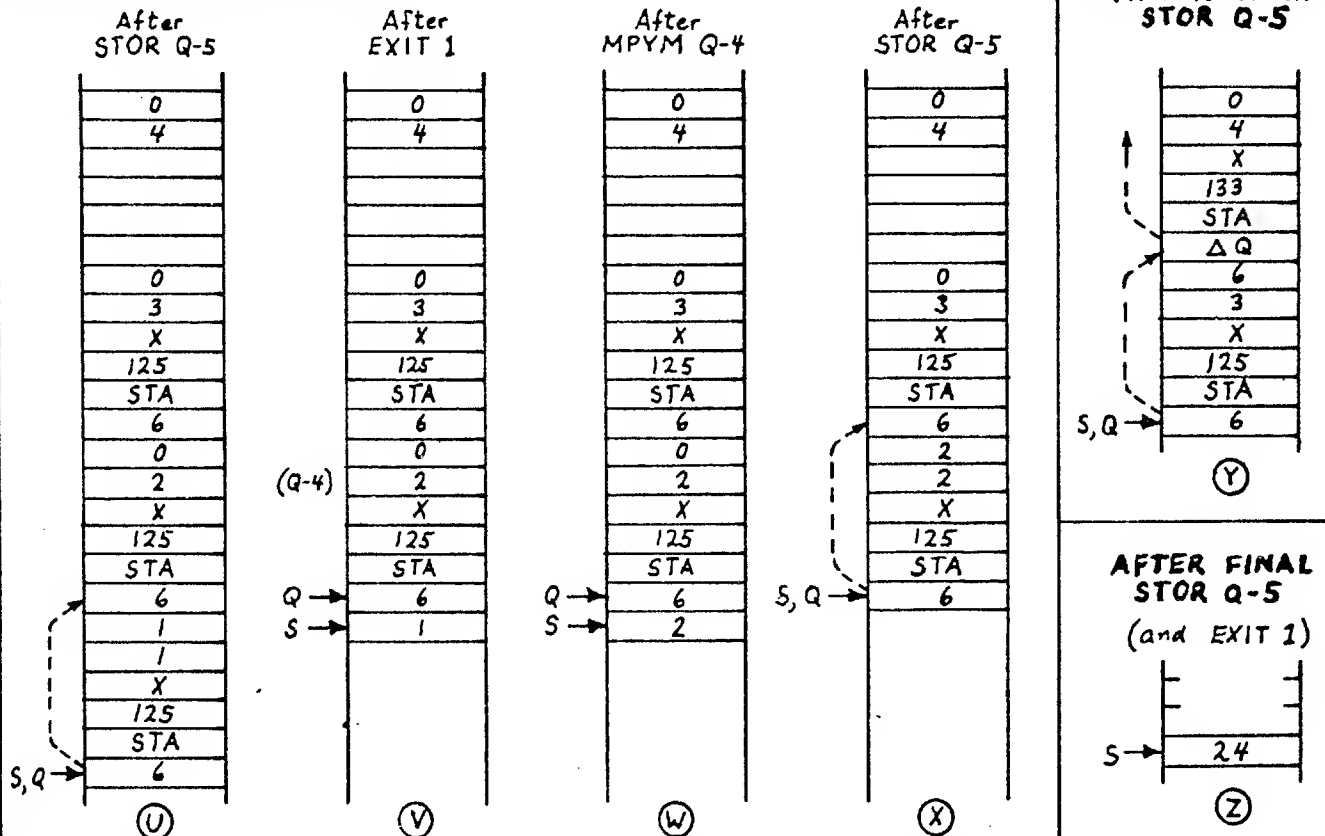


Figure 4-17. Recursive Exits

beginning with the parameter test for zero. Since the parameter is 3 on the second recursion, the branch to address 121 again occurs. The first actions, again, are to reserve a location for this recursion's answer (J) and to load a decremented parameter value of 2 (K and L). After this, the procedure call back to the beginning is again made, resulting in another stack marker (M) which is identical to the one generated on the first recursion.

The third and fourth recursions repeat the entire process again, loading parameters of 1 and 0 followed each time by a stack marker. Thus when the final LOAD Q-4 occurs in preparation for the zero test, the stack appears as shown at N.

FIRST EXIT. The check at address 115 now finds that the parameter is zero. The checked copy of the parameter is deleted from the stack (P in figure 4-17) and the branch at address 116 transfers control to address 117 (rather than 121).

As mentioned earlier (fourth paragraph under the Recursion heading), an assumed value of 1 is necessary as an initial "previous product" in order to begin the multiplication loops. This is accomplished by a LDI 1 instruction (address 117), which puts a 1 on the top of the stack (see Q).

Then an unconditional branch at address 120 transfers control to address 126, where the "1" on the top of the stack is stored into the location reserved for this recursion's answer, as shown at R. The next instruction is then the EXIT 1 instruction at address 127. This causes Q to move back six locations ($\Delta Q = 6$) and S five locations (EXIT 1 saves one parameter), as shown at S. The return address for the P-register, as will be remembered from five paragraphs back, is the MPYM Q-4 instruction at address 125. This causes the parameter at Q-4 (1) to be

multiplied by the 1 on the top of the stack, leaving the answer as the new top-of-stack element. Since $1 \times 1 = 1$ there is no apparent change from S to T, but in fact a multiplication has occurred.

FIRST RECURSIVE EXIT. The answer of the first multiplication is now stored in the location reserved for it (Q-5) as shown at U, by the STOR Q-5 instruction at address 126. The next instruction, at 127, is again the EXIT 1 instruction, which peels back the stack as shown at V and returns the P-register to the MPYM Q-4 instruction at address 125. The parameter for multiplication (at Q-4) is now 2, so the multiplication result at W is 2. Again, this is stored back in the location reserved for it (Q-5) as shown at X.

SUCCESSIVE EXITS. After saving the result, the next EXIT 1 is again encountered, causing the S and Q stack pointers to move back to the next marker, leaving the answer 2 on the top of the stack. The return for the P-register is again 125, so the MPYM Q-4 instruction multiplies 2×3 , and the following STOR Q-5 puts the answer 6 into the reserved location as shown at Y.

Likewise, the last recursive exit causes the value 6 to be left on the top of the stack when the last return to address 125 is made. Then the final multiplication multiplies 6×4 , and the last STOR Q-5 instruction puts the answer 24 into the location originally reserved for the end result FACTORIAL.

The last EXIT instruction finds the return for the Q-register (ΔQ) pointing back to the origin of an earlier procedure, and so is no longer shown in the stack diagram at Z. However, since one parameter is saved, the final answer remains on the top of the stack, as shown. The P-register, meanwhile, returns to the next instruction in the outer block, which is the STOR DB 15 instruction at address 133. This saves the answer in the global area, and a final PCAL returns control to the system.